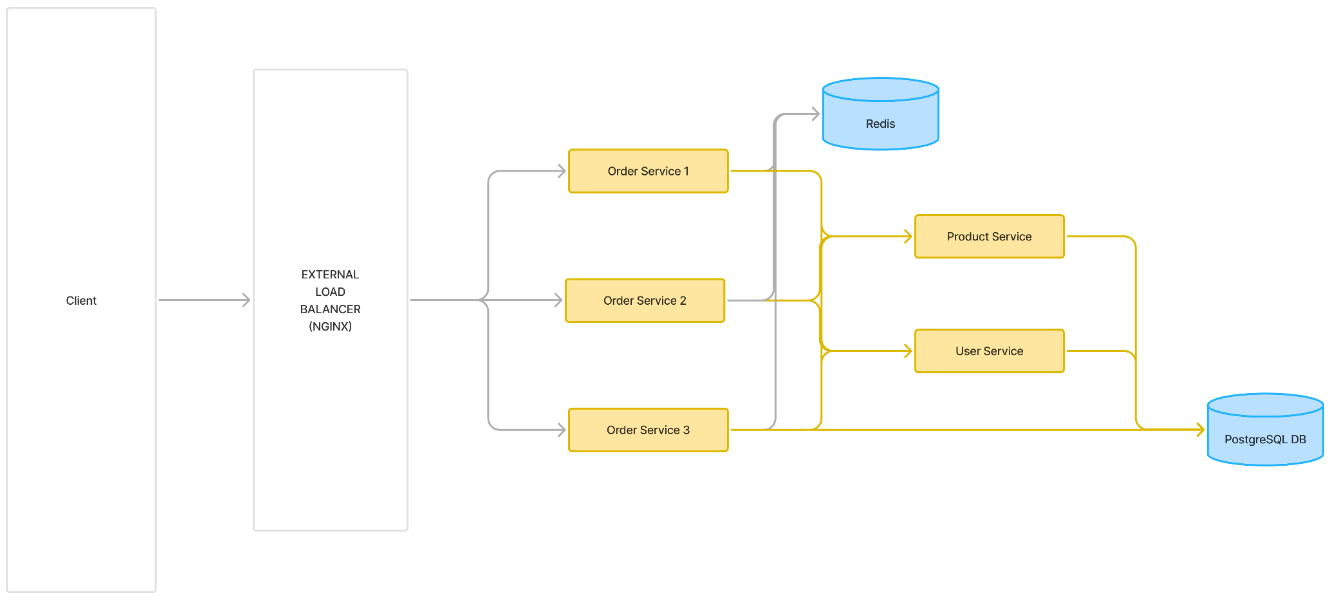
System Design

Last updated: March 19, 2025

Hyeonbin Chun

Software Architecture Diagram



Architecture: Microservices (Java Server) with Docker

- All services run as containers using Docker, ensuring scalability

Workflow:

1. The client sends requests to the External Load Balancer (Nginx), which acts as the single entry point for all incoming traffic.
2. The load balancer distributes requests across multiple instances of the Order Service, ensuring high availability and optimal performance.
3. Order Service Processing:
    a. Checks Redis Cache for relevant data (order, user, product) to reduce database load.
    b. If data is not found in Redis, call User Service or Product Service to retrieve data from postgreSQL.
    c. Stores the retrieved data in Redis for future requests, improving response time and efficiency.

Profiling & Justification

***Average latency** refers to the average amount of time it takes for the server to respond to a request. This time includes everything from the moment you send the request to the moment you receive the response. (Sum of all request-response times / Total number of requests)

***Actual RPS** refers to the actual requests per seconds measures by (total requests / duration of test)

***Latency threshold**: 0.1s

Note: mixed request (order, user, product) with 30s duration time is used for each RPS testing.

Baseline Test: Measure initial system performance before optimizations.

Testing 5 RPS	Testing 10 RPS	Testing 20 RPS	Testing 40 RPS	Testing 60 RPS	Testing 65 RPS
Actual RPS: 5.60 Average latency: 0.0634	Actual RPS: 10.50 Average latency: 0.0594	Actual RPS: 21.00 Average latency: 0.0929	Actual RPS: 42.00 Average latency: 0.1225	Actual RPS: 55.50 Average latency: 0.1818	Actual RPS: 72.90 Average latency: 0.6957

Result: Server can handle approximately 20 requests per second.

First Optimization:

1. Eliminates ISCS: OrderService directly sends requests to the User Service and Product Service.
2. Eliminates duplicate codes / functions (clean-up OrderService code)

Testing 5 RPS	Testing 10 RPS	Testing 20 RPS	Testing 40 RPS	Testing 80 RPS	Testing 160 RPS	Testing 320 RPS	Testing 500 RPS
Actual RPS: 5.60 Average latency: 0.0329 seconds	Actual RPS: 10.50 Average latency: 0.0420 seconds	Actual RPS: 21.00 Average latency: 0.0598 seconds	Actual RPS: 42.00 Average latency: 0.0747 seconds	Actual RPS: 84.00 Average latency: 0.1121 seconds	Actual RPS: 168.00 Average latency: 0.2338 seconds	Actual RPS: 339.60 Average latency: 0.2740 seconds	Actual RPS: 509.40 Average latency: 0.6758 seconds

Result: Server can handle approximately 50 requests per second.

- Reduced network overhead by eliminating unnecessary service hops.

- Streamlined code execution, making OrderService more efficient.
- Server capacity increased from 20 RPS → 50 RPS.
- Lower average latency across all tests.

Second Optimization:

1. Containerized all services (UserService, ProductService, OrderService instances).
2. Implemented NGINX as a load balancer to distribute requests evenly.

Testing 5 RPS	Testing 10 RPS	Testing 20 RPS	Testing 40 RPS	Testing 80 RPS	Testing 160 RPS	Testing 320 RPS	Testing 500 RPS
Actual RPS: 5.60 Average latency: 0.0390 seconds	Actual RPS: 10.50 Average latency: 0.0476 seconds	Actual RPS: 21.00 Average latency: 0.0626 seconds	Actual RPS: 42.00 Average latency: 0.0710 seconds	Actual RPS: 52.50 Average latency: 0.0585 seconds	Actual RPS: 168.00 Average latency: 0.0435 seconds	Actual RPS: 336.00 Average latency: 0.1210 seconds	Actual RPS: 529.50 Average latency: 0.1122 seconds

Result: Server can handle approximately 300 ~ 400 requests per second.

- Load balancing prevented a single OrderService instance from becoming overloaded.
- Scalability improved, as multiple service instances could handle requests in parallel.
- Server capacity jumped from 50 RPS → 300~400 RPS.
- Latency remained low even as RPS increased.

Third Optimization:

1. Change text file based database to PostgreSQL database.

Testing 320 RPS	Testing 640 RPS	Testing 750 RPS	Testing 1000 RPS
Actual RPS: 256.10 Average latency: 0.1072 seconds	Actual RPS: 233.40 Average latency: 0.1656 seconds	Actual RPS: 139.37 Average latency: 0.2633 seconds	Actual RPS: 8.27 Average latency: 4.8782 seconds

Result: Server can handle approximately 200 ~ 300 requests per second.

- Increased data reliability and support for complex queries. However, database response time increased, since PostgreSQL required proper indexing and optimization.

Fourth Optimization:

1. Implemented HikariCP (Database Connection Pooling).
2. Added database indexes to improve query speed.
3. Optimized SQL queries for better performance.

Testing 320 RPS	Testing 640 RPS	Testing 750 RPS	Testing 1000 RPS
Actual RPS: 320.17 Average latency: 0.0133 seconds Status codes: {200: 9605}	Actual RPS: 622.83 Average latency: 0.0314 seconds Status codes: {200: 18537, 400: 4, 0: 144}	Actual RPS: 635.17 Average latency: 0.0256 seconds Status codes: {0: 10507, 200: 8547, 400: 1}	Actual RPS: 821.30 Average latency: 0.0311 seconds

Result: Server can handle approximately 550 ~ 650 requests per second.

- HikariCP reduced connection overhead, avoiding the cost of opening/closing DB connections repeatedly.
- Using index and proper SQL query optimize the database querying time
- Server capacity increased to 550~650 RPS.
- Latency dropped significantly.
- Faster database queries & optimized connection handling led to higher throughput.

Fifth Optimization:

1. Integrated Redis caching for:
   a. User existence checks
   b. Product existence checks
   c. Product stock availability checks

Testing 320 RPS	Testing 640 RPS	Testing 750 RPS	Testing 1000 RPS
Actual RPS: 313.57 Average latency: 0.0192 seconds Status codes: {200:	Actual RPS: 604.20 Average latency: 0.0164 seconds Status codes: {200:	Actual RPS: 441.97 Average latency: 0.0434 seconds Status codes: {0:	Actual RPS: 11.70 Average latency: 3.4320 seconds Status codes: {200:

9407}	16012, 400: 1, 0: 2113}	2225, 200: 11034}	111, 0: 240}

Result: Server can handle approximately 550 ~ 650 requests per second.

- The results are similar to fourth optimization, suggesting we might need a better Redis Cache strategy or better Redis configuration.